Presentation Services

- need for a presentation services
- ASN.1
  - declaring data type
  - encoding data types
- implementation issues
- reading: Tannenbaum 7.3.2

Presentation Services: Motivation

**Question:** suppose we could copy reliably from one computer’s memory to another. Would this “solve” communication problem?

**Answer:** ?

**Crux of problem:**
- deal with *meaning* of information, not *representation*
- different computers, OS, compilers have different conventions for representing data
  - architecture: big endian versus little endian
  - floating point format
  - data type size: 16, 32, 64 bit int
  - different size, layout of data structures
Solving the representation problem

- have sender encode to receiver’s format
- have receiver decode from sender’s format
- have machine-, OS-, language-independent method for describing data structures
  - host translates to/from universal description language from/to own format
- pros and cons?

ASN.1: Abstract Syntax Notation 1

ISO standard (one still meaningful)

**abstract syntax:** “language” for describing data structures
- data description language, not programming language
- defines universal data types
- allows user-defined data types

**basic encoding rules:**
- convert abstract syntax specification of data structure into series of bytes (for transmission)
### ASN.1: a pictorial view

Module of data type declarations written in ASN.1 abstract syntax:

- `lastname ::= OCTETSTRING`
- `weight ::= INTEGER`

Instances of data types defined in "module":

- `{weight,259}`
- `{lastname,"Smith"}`

Basic encoding rules (BER):

- Octets of encoded data:
  - 1: height
  - 2: tim
  - 3: S
  - 4: t
  - 5: 5
  - 6: 3

---

### ASN.1: Universal Types

Predefined types with given tag value:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BOOLEAN</td>
<td>value is true or false</td>
</tr>
<tr>
<td>2</td>
<td>INTEGER</td>
<td>can be arbitrarily big</td>
</tr>
<tr>
<td>3</td>
<td>BITSTRING</td>
<td>list of one or more bits</td>
</tr>
<tr>
<td>4</td>
<td>OCTET STRING</td>
<td>list of one or more bytes</td>
</tr>
<tr>
<td>5</td>
<td>NULL</td>
<td>no value</td>
</tr>
<tr>
<td>6</td>
<td>OBJECT IDENTIFIER</td>
<td>refers to an &quot;object&quot;, e.g. protocol number</td>
</tr>
<tr>
<td>9</td>
<td>REAL</td>
<td>floating point</td>
</tr>
</tbody>
</table>

Example declarations: think of ::= as defining new data type in terms of universal data type

- `Married ::= BOOLEAN`  
- `SSN ::= INTEGER`  
- `Lname ::= OCTETSTRING`  
- `Salary ::= REAL`  
- `IPAddress ::= OCTETSTRING (SIZE 4)`
**ASN.1 Syntax: constructors**

ASN.1 defines constructor types for building more complex data types of “simpler” data types:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>SEQUENCE</td>
<td>ordered list, each element an ASN.1 type</td>
</tr>
<tr>
<td>17</td>
<td>SET</td>
<td>same as sequence but unordered</td>
</tr>
<tr>
<td>11</td>
<td>CHOICE</td>
<td>a type taken from specified list</td>
</tr>
</tbody>
</table>

Example of constructed data type:

```plaintext
studentRecord ::= SEQUENCE { 
  Lname OCTETSTRING,  
  Fname OCTETSTRING,  
  Mname OCTETSTRING,  
  Married BOOLEAN DEFAULT FALSE,  
  SSN INTEGER }
```

**ASN.1 Encoding Example**

The ASN.1 definition:

```plaintext
Attendee ::= SEQUENCE { 
  name OCTET STRING,  
  paid BOOLEAN }
```

The data (“Smith”, `true`) would be encoded:

```
<table>
<thead>
<tr>
<th>Sequence (10 bytes long)</th>
<th>OCTET STRING (5 bytes long)</th>
<th>BOOLEAN (1 byte long)</th>
</tr>
</thead>
</table>
```

Note nesting of TLV structure in above example.
**ASN.1: But how do I use it?**

Normal people don’t want to write encoding/decoding routines!

ASN.1 “compilers” take ASN.1 abstract syntax module and produce
- C data type definitions (e.g., typedef’s) that user can #include to create data structures having these types
- library of C-callable routines (e.g., one for each data type) to encode/decode each typedef to/from TLV encoding

---

**External Data Representation: XDR**

- developed by SUN (RFC 1014)
- similar to ASN.1 in power
- the de facto standard for most client-server applications
  - underlies SUN RPC and NFS
- both stream oriented (TCP) and record oriented (UDP)
- XDR can be combined with remote procedure calls
  - rpcgen compiler allows you to write rpc and encodes data in XDR format
Presentation Services: closing thoughts

- presentation processing expensive:
  - up to 90% processing time on ethernet/IP/TCP/presentation stack
  - cost to encode array of int's 5-20 times more expensive than copy
  - too heavyweight?

- interesting reading:
  - John Larmouth's book "Understanding OSI": chapter 8: ASN.1
  - role of ASN.1 in next generation http

Network Application Programming

**Introduction:** issues

**Sockets:** programming and implementation

**Other API's:**
- winsock
- java
- transport layer interface (TLI)
- Novell netware API

**Reading:** Tannenbaum, page 486-487, KR Chapter 2
ftp://gaia.cs.umass.edu/cs653/sock.ps
The Application Programming Interface: API

- **API**: the programming model, application callable services, interfaces, and abstractions provided by the network (i.e., lower layers) to the application.
- does an API provide for:
  - *naming and service location*: must application know precise location (e.g., host address and port) of service? Can services be requested by name? Can servers registers services?
  - *connection management*: must applications do low-level handshaking required to setup/teardown connection?

The API (continued)

Does an API provide for:
- **message transfer**
  - application-selectable data transfer services: best-effort versus reliable?
  - message priorities?
  - multi-site atomic actions?
  - structured versus byte-stream communication?
- **communication flexibility**
  - can application select and/or modify protocol stacks (statically or dynamically)?
- **Quality of Service specification**
  - can application specify QoS requirements to network?
The SOCKET API

- introduced in 1981 BSD 4.1 UNIX
- a **host-local, application created/owned, OS-controlled interface** into which application process can both **send and receive messages** to/from another (remote or local) application process

![Diagram of socket API](image)

The SOCKET API (cont)

- two sockets on separate hosts "connected" by OS socket management routines. Application only sees local socket.
- sockets explicitly created, used, released by applications
- based on client/server paradigm
- two types of transport service via socket API:
  - unreliable datagram
  - reliable, stream-oriented
- presentation, session layers missing in UNIX networking (an application concern!)
Sockets: conceptual view

- each socket has separate send/receive buffers, port id, parameters (application queryable and setable).
- socket operations implemented as system calls into OS
- user/kernel boundary crossed: overhead
Connectionless Service

- **datagram service**: underlying transport protocols do not guarantee delivery
- no explicit identification of who is server, who is client
- if initiating contact with other side, need to know
  - IP address
  - port number of process waiting to be contacted.
- if waiting for contact from other side, need to declare
  - port number at which waiting for other side

---

**SERVER**
1. create transport endpoint: `socket()`
2. assign transport endpoint an address: `bind()`
3. wait for pkt to arrive: `recvfrom()`
4. send reply (if any): `sendto()`
5. release transport endpoint: `close()`

---

**CLIENT**
1. create transport endpoint: `socket()`
2. assign transport endpoint address: (optional) `bind()`
3. determine address of server
4. send msg: `sendto()`
5. wait for pkt to arrive: `recvfrom()`
6. Release transport endpoint: `close()`
DNS: Internet Domain Name System

- a distributed database used by TCP/IP applications to map to/from hostnames from/to IP addresses
- name servers:
  - user-level library routines gethostbyname() and gethostbyaddr() contact local nameserver via port 53
  - name server returns IP address of requested hostname

Diagram:
- Application invokes gethostbyname() or gethostbyaddr() to contact local nameserver via port 53.
- Name server returns IP address of requested hostname.
**DNS: non-local names**

**finding non-local names**
- no single name server has complete info
- if local name server can't resolve address, contacts root name server:
  - 9 redundant root nameservers world-wide
  - each has addresses of names servers for all level-two name servers (e.g., umass.edu, ibm.com)
  - contacted root server returns IP address of name server resolver should contact
  - contacted level-two name server may itself return a pointer to another name server
  - name resolution an iterative process of following name server pointers
  - DNS protocol specifies packet formats for exchanges with DNS servers

---

**Assigning socket a network address: bind()**

- each socket must be associated with a local, host-unique 16-bit port number.
- need to associate socket with globally unique network address (host address and port)
  - OS knows that incoming messages addressed to this host address and port to be delivered (demultiplexed to) to this socket
  - a return address for outgoing messages
## Port Numbers

<table>
<thead>
<tr>
<th>Port number(s)</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 - 255</td>
<td>reserved for standard services</td>
</tr>
<tr>
<td>21</td>
<td>ftp service</td>
</tr>
<tr>
<td>23</td>
<td>telnet service</td>
</tr>
<tr>
<td>25</td>
<td>SMTP email</td>
</tr>
<tr>
<td>80</td>
<td>http daemon</td>
</tr>
<tr>
<td>1 - 1023</td>
<td>available only to privileged users</td>
</tr>
<tr>
<td>1024 - 4999</td>
<td>usable by system and user processes</td>
</tr>
<tr>
<td>5000 -</td>
<td>usable only by user processes</td>
</tr>
</tbody>
</table>

## Connection-oriented service
**Connection-oriented service**

- **client/server handshaking:**
  - client must explicitly connect to server before sending or receiving data
  - client will not pass `connect()` until server accepts client
  - server must explicitly accept client before sending or receiving data
  - server will not pass `accept()` until client `connect()`’s
- connection-oriented service: underlying transport service is **reliable, stream-oriented.**
**Typical server structure**

- `socket()`
- `bind()`
- `listen()`
- `accept()` to create a child process, `fork()` to handle communication (provide service) to client
- `sendto()`, `recvfrom()` to communicate with client and provide service via new socket
- `close(newsockid)` and `exit()`

**Aside: other useful system calls and routines**

- `close(sockfd)` will release a socket
- `getsockopt()` and `setsockopt()` system calls used to query/set socket options.
- `ioctl()` system call used to query/set socket attributes, also network device interface attributes.
Implementation: OS actions on `sendto()`

- `sendto()` system call causes interrupt
- check high-water mark: enough space for user data?
- copy data from user's address space into socket buffers (kernel space)
- procedure call down to transport layer (e.g., `tcpsend()`) to send data
- return from system call
- more data to send?
- wait until buffer space frees up
- return from procedure call to transport layer
- socket identifier different from file identifier
- read(), write(), close() should not be used
- uses socket-specific equivalents instead

Windows Sockets

Based on BSD sockets:
- BSD: ``the de facto standard for TCP/IP Networking''
  (quote from Winsock1.1 documentation)
- supports stream(TCP)/datagram(UDP) model
- API the same as what we have seen

A few differences/incompatibilities:
- extensions for asynchronous programming
- different error return codes: -1 not the error return code!
- socket identifier different from file identifier
- read(), write(), close() should not be used
- use socket-specific equivalents instead
API: Summary

- some API’s provide only low-level interface to transport services: socket, winsock, TLI
- other API’s provide higher-level services (e.g., transaction support, service advertising or request)
  - makes building applications easier
- sockets the de facto standard
- FYI reading:
  - winsock: http://www.sockets.com
  - JAVA: http://java.sun.com
  - Tutorial on sockets: http://manic.cs.umass.edu